

2009

PACIFIC NW SOFTWARE  
QUALITY  
CONFERENCE



MOVING  
QUALITY  
FORWARD

OCTOBER 27-28, 2009

Conference Paper Excerpt

From the

CONFERENCE  
PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

# Web Security Testing with Ruby and Watir

James O. Knowlton  
james\_knowlton@mcafee.com

## Abstract

To verify the quality of web applications today, security testing is a necessity. But how to cover it all? SQL injection, cross-site scripting, buffer overflow...and the list goes on. Automating some of this testing would be great, but where to start?

This paper is a case study in how McAfee decided to use Ruby/Watir to help with it's Web security testing needs

The Ruby language, combined with the Watir module, is a great toolset for security testing of web applications. There are three reasons for this:

- Many of the common security vulnerabilities related to web applications (SQL Injection, cross-site scripting, and buffer overflow) have to do with simply posting different types of information to a web server via a client. This is pretty much what Watir is all about. It even gives you access to hidden elements, so its really is a great tool for submitting form data to a web server.
- The Ruby side of Watir, being a full-service language, has great tools for querying the database, checking audit logs and the like. Also, you can generate random data (or large datasets) to throw at a web app, or even pull the test data from a CSV file.
- There are some things you might not be able to do through Watir, but can certainly be done with Ruby. Again, this is perfect – because Watir is not really a test framework, it's just a way to drive the Browser when you need to. So, tests which are more low-level (such as web service communication or network tests) can be run through Ruby and RSpec, or whatever actual test framework you're using.

## Biography

Jim Knowlton is a QA Automation Engineer with McAfee Inc., where he drives test automation efforts on McAfee ePolicy Orchestrator. He has over 18 years' experience in the software industry, including clients such as Novell, Symantec and Nike. He has been a Technical Support Engineer, a Technical Writer, a Beta Program Manager, and some other jobs he can't remember. He has a Bachelor's Degree in Management and is currently pursuing a Master of Business Administration degree at the University of Portland.

Jim is the author of "Python: Create, Modify, Reuse" (Wrox, 2008), and has also written website technical content for America Online, Amazon and IBM. He blogs at [www.agilerubytester.com](http://www.agilerubytester.com).

# Introduction

We're going to talk about using Ruby, along with Watir, in building a security testing framework for web applications.

This paper will consist of the following:

1. A description of the problem we were trying to solve at McAfee
2. The possible choices we had and why we chose to implement our tests with Ruby/Watir
3. A quick overview of Ruby and Watir, and why we felt it would be especially good for what we were trying to accomplish
4. Demo of a few running examples of cross-site scripting and SQL injection tests running against my blog.
5. A review of our experience implementing the Ruby/Watir test framework
6. Lessons learned from our experience

## Some security terms

To understand our challenge and how we approached building a solution, it is important to understand some basic types of security attacks applied to a web application:

- SQL injection - A SQL injection attack is one where an attacker takes advantage of the fact that information is passed from the web server to the database, and that the web server has access to the database. The attacker passes SQL commands in the input fields of a web application, and if the attack is successful, those SQL commands are then executed on the database server.
- Cross-site scripting - A cross-site scripting attack is one where a user embeds JavaScript in information submitted to the web server. That JavaScript would then be stored, and if another user brought the page up with that saved information, the JavaScript would then be executed. This can allow an attacker to get another user to go to a different website, reveal their username or password, or any number of other activities.
- Buffer overflow - A buffer overflow is simply an attempt to overflow the ability of the web server (or database) to process information. It consists of sending a huge amount of data to the web server. Often the goal of a buffer overflow attack is not to compromise a system, but to create a denial of service (make the system no longer available to others).

## The challenge – testing security of a web application

McAfee has a web-based security management platform called ePolicy Orchestrator. It allows users to log in and perform various operations on client machines in a network, and run reports on the state of those machines.

Our challenge was to execute security tests (especially SQL injection and cross-site scripting tests) in an automated fashion, on all builds delivered to QA. As we saw it, we had the following requirements:

- The tests needed to be *automated* – we needed to be able to “push a button” and run all the tests, or subsets of the tests, and get results, with as little user interaction as possible
- The tests needed to be *easily developed and understood* – we wanted to develop the tests in a language and infrastructure that would be easy to understand and learn, since we would possibly be handing our tests off to other groups to run.
- The tests needed to be *license-free* – since we would be handing the tests off to other groups, and we had no idea if they would have licenses for any licensed technologies, we wanted to use open-source technologies to insure anyone would be able to set up a test environment.
- The tests needed to be *in a widely supported environment* – since development resources for support would be very limited, we needed to choose technologies for our test environment that are widely used and that have active communities (newsgroups, web sites, etc), so that we would have places to go should we run into issues (which we certainly would).

## Choices and possible solutions

The following are two possible solutions we looked at, but decided not to pick.

### Java/JUnit/Ant/WebTest

Since our application is written in Java, this is the first solution that occurred to us.

#### Advantages:

- Developers could give advice, since they all know Java
- Could directly access product classes
- Stable, mature architecture
- Development tool support - IntelliJ (IDE) and TeamCity (build automation tool) both support Java fully
- Canoo WebTest - We knew we would, for some tests, need something to drive the browser. WebTest is a good tool with an active community.

#### Disadvantages:

- Not much Java knowledge inside QA
- Since it is the same language as the application, there might sometimes be temptations to "hook into" the code and mock things when we really should be testing at the application level
- Building Java-based test frameworks tend to be more complex (sometimes necessarily so) than a scripting-language-based framework.

### Python/unittest/batch files/PAMIE

Another option was to build a framework with Python, a popular scripting language. We broke down this option thus:

#### Advantages:

- Scripting language - Python is certainly easier to learn to be productive in than Java, for QA engineers who are not programmer-types
- Mature - Python has been around a long time, and there are tons of libraries and modules for it
- I'm very familiar with it - I've written a book on Python, so obviously I like it and am pretty familiar with it.

#### Disadvantages:

- PAMIE - PAMIE, which is a Python-based tool to drive internet explorer, is not often updated and lags behind open-source competitors in features.
- No build configuration tool - there really isn't a Python-based build configuration tool that is as mature and "used" as Ant (Java) or Rake (Ruby).
- Little integration with existing tools - IntelliJ has a Python module, but it's fairly basic, and there really is no Python support in TeamCity.

## Ruby/Watir – why we picked it

We decided to go with a framework based on Ruby, RSpec, Rake and Watir. Why? Let's look at each technology in turn.

## Ruby

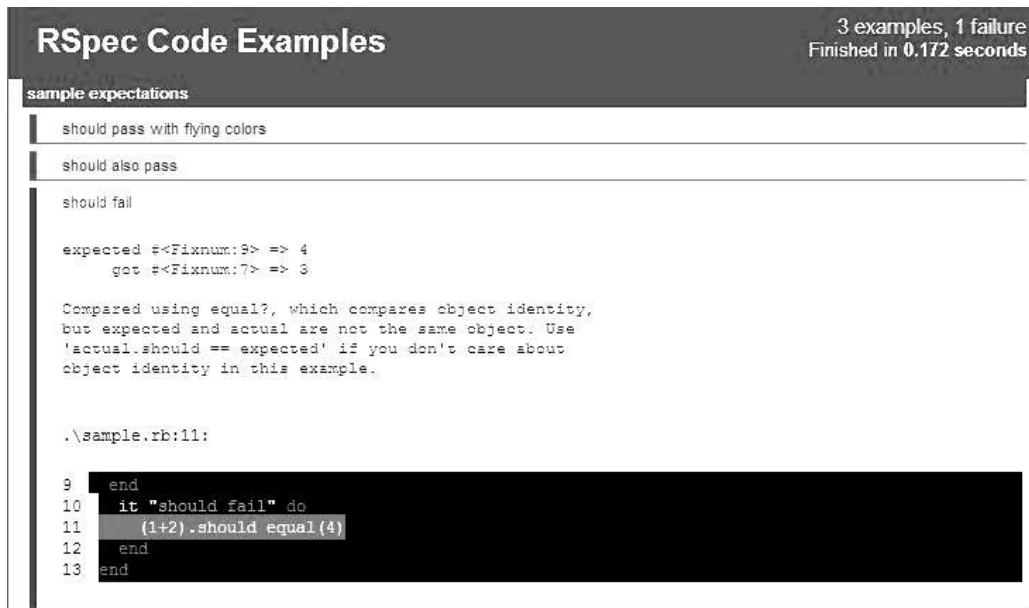
Ruby is an object-oriented, open-source language that is very popular in the testing community. It is pretty easy to learn and there are a multitude of books, websites and online communities devoted to it.

Another advantage of Ruby is that our IDE, JetBrains IntelliJ, has a great Ruby module with full support for Ruby, RSpec and Rake.

## RSpec

RSpec is a Ruby-based test framework. I chose RSpec for two main reasons:

- Test names can be written out in english, making test reports much easier to read
- RSpec can produce a really nice HTML test report:



The screenshot shows an RSpec HTML test report titled "RSpec Code Examples". In the top right corner, it displays "3 examples, 1 failure" and "Finished in 0.172 seconds". The report content is as follows:

```
sample expectations
  should pass with flying colors
  should also pass
  should fail
    expected =<Fixnum:9> => 4
      got =<Fixnum:7> => 3

    Compared using equal?, which compares object identity,
    but expected and actual are not the same object. Use
    'actual.should == expected' if you don't care about
    object identity in this example.

    .\sample.rb:11:
9     end
10    it "should fail" do
11      (1+2).should equal(4)
12    end
13  end
```

## Rake

Rake is the build configuration tool for Ruby. It has the following features:

- Rakefiles are actually Ruby scripts, so you have the full power of Ruby within your rakefile
- Rake has targets for RSpec tests, so it was easy to write test targets in Rake for RSpec
- Rake is fully supported in JetBrains TeamCity, our build automation tool.

## Watir

We knew that for some of our tests we would need a way to drive the browser (there weren't API hooks in place to access product functionality). The advantage of using a COM-based tool which actually drives the real browser, rather than simulating a browser (as a tool like Canoo Webtest does) is that anything the browser can do, you can do automatically. Thus, it fully supports Ajax and Javascript, whereas WebTest "mostly" provides support.

Watir is the most mature and stable of the COM-based IE drivers, and it has a significant community behind it.

# Some example security testing scripts

## SQL injection

The following script is an example of a script to test for SQL injection. SQL injection is an attack technique that consists of embedding SQL commands in the entry fields of a web page. So, the basic way to test for this is to embed SQL commands in the web page to submit, then connect to the database and verify that the SQL command did not get executed. See the comments below (lines preceded by a '#' to follow the flow of the script:

```
#require needed modules
require 'spec'
require 'watir'
require 'dbi'

describe 'SQL Injection' do
  #initialize required variables
  before(:all) do
    username="testuser"
    password="testpassword"
    url="http://www.mywebapplication.com/login"
  end
  #first testcase, for sql injection, command to drop a table
  it 'should not be allowed when entering login name - drop table' do
    #start IE and go to login page
    ie = Watir::IE.new()
    ie.goto(url)
    #enter SQL command for login name and submit
    ie.text_field(:id, "username").set("'drop table reallyImportant'")
    ie.text_field(:id, "password").set(password)
    ie.button("Log On").click()

    #connect to db and check that table was not dropped
    connectstring="DBI::ODBC::mydbserver"
    db=dbi.connect(connectstring, "sa", "dbpassword")
    tablelist = db.tables #returns a list of db tables
    tablelist.should include("reallyImportant")
    db.disconnect()
  end
  #second testcase, for sql injection, command to create a table
  it 'should not be allowed when entering login name - create table' do
    #start IE and go to login page
    ie = Watir::IE.new()
    ie.goto(url)
    #enter SQL command for login name and submit
    ie.text_field(:id, "username").set("'create table junk\(varchar 20\)\)")
    ie.text_field(:id, "password").set(password)
    ie.button("Log On").click()

    #connect to db and check that table was not dropped
    connectstring="DBI::ODBC::mydbserver"
    db=dbi.connect(connectstring, "sa", "dbpassword")
    tablelist = db.tables #returns a list of db tables
    tablelist.should_not include("junk")
    db.disconnect()
  end
end
end
```

## Cross-site scripting

The following script is an example of a script to test for cross-site scripting. Cross-site scripting is an attack technique that consists of embedding JavaScript commands in the entry fields of a web page. So, the basic way to test for this is to embed JavaScript in the web page to submit (in this case JavaScript to generate a popup), then verify the JavaScript was not executed. Note that in the example below, I am assuming there is a `check_for_popups()` method - to actually check for Windows popups, take a look at the Win32OLE/AutoIT module in Ruby documentation. See the comments below (lines preceded by a '#') to follow the flow of the script:

```
#require needed modules
require 'spec'
require 'watir'
require 'dbi'

describe Cross Site Scripting' do
  # initialize required variables
  before(:all) do
    username="testuser"
    password="testpassword"
    url="http://www.mywebapplication.com/login"
  end
  #first testcase, for XSS string 1
  it 'should not be allowed when entering login name - XSS string 1' do
    #start IE and go to login page
    ie = Watir::IE.new()
    ie.goto(url)
    #enter XSS string for login name and submit
    ie.text_field(:id, "username").set("<script>alert('XSS');</script>")
    ie.text_field(:id, "password").set(password)
    ie.button("Log On").click()

    #use WIN32OLE/AutoIt to check for Windows popups
    check_for_popups("Windows Internet Explorer").should == false
  end
  #second testcase, for XSS string 2
  it 'should not be allowed when entering login name - XSS string 2' do
    #start IE and go to login page
    ie = Watir::IE.new()
    ie.goto(url)
    #enter XSS string for login name and submit
    ie.text_field(:id, "username").set("<BODY ONLOAD=alert('XSS')>")
    ie.text_field(:id, "password").set(password)
    ie.button("Log On").click()

    #use WIN32OLE/AutoIt to check for Windows popups
    check_for_popups("Windows Internet Explorer").should == false
  end
end
end
```

## These scripts could be improved...

- Move common functionality into helper modules
- Extract data out and put in yaml (properties) file

## Other types of tests

You could also use this framework for other types of web security tests (as we have), such as:

- Buffer overflow - Enter large amounts of data in form fields, and validate you get an appropriate error and no inappropriate errors on the server.

- URL manipulation - Have a list of product URLs. Feed the URLs one by one to a method that simply goes directly to the URL. Validate you get a login prompt.
- Information disclosure - Use Ruby directly to open log files and other system files; make sure there are no passwords or other sensitive information stored in plain text.

## Our experience

We are very happy with the choice we made of using Ruby in conjunction with other tools to build an automation framework for security tests.

We used these tests in conjunction with HP QAInspect, which does a much more exhaustive, "carpet bombing" test of a web application. We found that the two approaches were more complimentary than duplicative:

- The Ruby tests had the advantage of actually checking the database for SQL injections, whereas QAInspect merely inspects to see where there might be vulnerabilities to SQL injection.
- The Ruby tests tell us precisely what is being tested and what the response is, whereas QAInspect (at least out of the box) is not as precise.
- QAInspect can span the entire application in a way that scripting tests by hand could never approach.

## Some metrics

- Total security tests written - about 300
- Time to write the tests - about 1 month (we were working on other projects too, and it took a little longer than a month, so I'm estimating the "full time" time cost)
- Time it takes to run a full security test suite (all 300 tests) - a little over an hour

## Lessons learned

We feel like we learned the following lessons from this experience, which was mostly a positive one:

- Have a constant focus on what technology and process will produce the best results, the fastest. Always focus on your central mission in QA, which is to provide accurate and timely information on product quality. Sometimes we can fall in love with tools or technology and forget why we're using them.
- Be on the constant lookout for ways to improve your process, as well as technologies to make things better.
- Part of "agile testing" is taking the initiative to communicate constantly what you're doing and ask management "does this provide you with valuable information? If not, what would?" We got lots of good ideas for test development from developers.
- Don't ever be emotionally married to your creation. If tomorrow we found something that worked better for testing our web applications, then we'd drop what we have. Remember - it's a program, not a child.

## Final thoughts

We found Ruby, RSpec, Rake and Watir to be a great ecosystem for building web security tests (or any tests for a web application, for that matter). The combination of the power of the Ruby scripting language, the simplicity and cool HTML reports of RSpec, the flexibility of Rake and the ability of Watir to access the application directly through the browser makes this test solution flexible and powerful. Add in support for tools such as IntelliJ and TeamCity (Ruby is also supported very well in IDEs such as NetBeans and Eclipse), and you have a great tool for creating an effective automation framework.

# References

## Web Security

- Open Web Application Security Project (OWASP) - <http://www.owasp.org>
- SANS Institute - <http://www.sans.org>
- Foundstone WebSec101 online course - <http://www.foundstone.com/us/websec101.asp>

## Ruby

- Ruby website - <http://www.ruby-lang.org>
- 'Programming Ruby' book online (1st edition) - <http://www.rubycentral.com/book>
- Ruby Essentials website - [http://www.techotopia.com/index.php/Ruby\\_Essentials](http://www.techotopia.com/index.php/Ruby_Essentials)
- comp.lang.ruby newsgroup - <http://groups.google.com/group/comp.lang.ruby>

## RSpec

- RSpec website - <http://rspec.info>
- RSpec API documentation - <http://rspec.rubyforge.org/rspec/1.2.8>
- RSpec Google group (mirror of mailing list) - <http://groups.google.com/group/rspec>

## Watir

- Watir website - <http://www.watir.com>
- Watir API documentation - <http://wtr.rubyforge.org/rdoc>
- Watir-General Google group - <http://groups.google.com/group/watir-general>